Solving the Expression Problem in C++, à la LMS
(Author's Version) *

Seyed H. Haeri (Hossein)[1] and Paul Keir[2]

[1] Université catholique de Louvain, Louvain-la-Neuve, Belgium
hossein.haeri@uclouvain.be
[2] University of the West of Scotland, UK
paul.keir@uws.ac.uk

Abstract. We give a C++ solution to the Expression Problem that takes a components-for-cases approach. Our solution is a C++ transliteration of how Lightweight Modular Staging solves the Expression Problem. It, furthermore, gives a C++ encoding to object algebras and object algebra interfaces. We use our latter encoding by tying its recursive knot as in Datatypes à la Carte.

1 Introduction

The Expression Problem (EP) [6,31,37] is a recurrent problem in Programming Languages (PLs), for which a wide range of solutions have been proposed. Consider those of Torgersen [35], Odersky and Zenger [20], Swierstra [34], Oliveira and Cook [23], Bahr and Hvitved [2], Wang and Oliveira [38], Haeri and Schupp [16], and Haeri and Keir [12], to name a few. EP is recurrent because it is repeatedly faced over embedding DSLs – a task commonly taken in the PL community. Embedding a DSL is often practised in phases, each having its own Algebraic Datatype (ADT) and functions defined on it. For example, take the base and extension to be the type checking and the type erasure phases, respectively. One wants to avoid recompiling, manipulating, and duplicating one's type checker if type erasure adds more ADT cases or defines new functions on them.

Haeri [11] phrases EP as the challenge of implementing an ADT – defined by its cases and the functions on it – that:

E1. is *extensible in both dimensions*: Both new cases and functions can be added.
E2. provides *weak static type safety*: Applying a function f on a statically[3] constructed ADT term t should fail to compile when f does not cover all the cases in t.

* This work is partially funded by the LightKone European H2020 Project under Grant Agreement No. 732505 and partially by the Belgian National Fund for Scientific Research (F.R.S.-FNRS).

[3] If the guarantee was for dynamically constructed terms too, we would have called it strong static type safety.

E3. upon extension, forces *no manipulation or duplication* to the existing code.

E4. accommodates the extension with *separate compilation*: Compiling the extension imposes no requirement for repeating compilation or type checking of existing ADTs and functions on them. Compilation and type checking of the extension should not be deferred to the link or run time.

On the other hand, Rompf and Odersky [32] coin Lightweight Modular Staging (LMS) for Polymorphic Embedding [17] of DSLs in Scala. They employ a fruitful combination of the Scala features detailed in [21] that, as a side-product, offers a very simple yet effective solution to the EP. We call that side-product the "Scala LMS-EPS." In this paper, we offer a new C++ solution that is greatly inspired by the Scala LMS-EPS. We call our own solution the "C++ LMS-EPS."

Amongst the EP solutions, LMS is distinctive for its ease of extension: both in adding new ADT cases and functions defined on them. We chose to implement LMS in C++ to show the independence of LMS from Scala's combination of following features: `trait`s, abstract type members, and `super` calls. Instead, the C++ LMS-EPS makes use of the following C++ features: curiously recurring template pattern (§ 3.2), abbreviated function templates of C++20[4] (§ 3.3), user-defined deduction guides and variadic `template`s (§ 5.3), and, most notably, `std::variant` (§ 3.1). (For the unfamiliar reader, an introduction to those C++ features comes in Appendix A.) Unlike Scala, C++ is a mainstream language which is well-known for its efficiency. Similar to Scala, C++ is a multi-paradigm language with a high level of abstraction (from C++17 onward).

Given its presentation in C++, the C++ LMS-EPS machinery may look as an EP solution that is too specific to C++. In order to correct that impression, we recall that it is typical for EP solutions to be presented with tactful uses of a single language. Take Datatypes à la Carte [34], CDTs [1], PCDTs [2], and MRM [25] in Haskell, Polymorphic Variants [9] in OCaml, and LMS [32] and MVCs [22] in Scala. The C++ LMS-EPS is amongst the few EP solutions which are presented in a mainstream programming language.

Here is a list of our contributions:

The C++ LMS-EPS takes a *components-for-cases* (C4C) [11] approach (§ 2.1 and 3.1). It implements ADTs (§ 3.2) using an encoding of object algebra interfaces [26] that is akin to Swierstra's sum of functors [34]. We tie the recursive knot using F-Bounding [4]. To implement functions on ADTs (§ 3.3), the C++ LMS-EPS gets a simulation of Haskell's Combinator Pattern [28, §16] (§ 5.3) to first acquire an encoding of object algebras. Our latter encoding, however, does not use *self*-references [27]. The C++ LMS-EPS outperforms its Scala predecessor by ensuring strong static type safety (§ 4.2). The way to distinguish between the C++ LMS-EPS and EP solutions that use Generalised Algebraic Datatypes (GADTs) is in § 6. Detailed discussion on the related work comes in § 7.

[4] Although our codebase remains fully functional without that (using ordinary type parametrisation), we retain its usage here for enhanced readability.

2 Background

2.1 Formal Notation

In this paper, we use parts of the $\gamma\Phi C_0$ calculus developed for solving the Expression Compatibility Problem [15]. $\gamma\Phi C_0$ was developed after observing that sharing ADT cases amongst ADTs is not limited to ADTs only extending one another. For example, consider the ADTs α_1, α_2, and α_3 defined as: $\alpha_1 ::= Num(\mathbb{Z}) \mid Add(\alpha_1)$, $\alpha_2 ::= Num(\mathbb{Z}) \mid Add(\alpha_2) \mid Mul(\alpha_2)$, and $\alpha_3 ::= Num(\mathbb{Z}) \mid Add(\alpha_3) \mid Sub(\alpha_3)$. Both α_2 and α_3 extend α_1. But, neither of them is an extension to the other. In order to share the implementation effort required for encoding α_2 and α_3, then $\gamma\Phi C_0$ promotes the ADT cases to *components* (in their Component-Based Software Engineering [33, §17],[29, §10] sense).

In $\gamma\Phi C_0$, ADT cases are independent of ADTs but still parameterised by them. In the $\gamma\Phi C_0$ notation, one would write $\alpha_1 = Num \oplus Add$, $\alpha_2 = Num \oplus Add \oplus Mul$, and $\alpha_3 = Num \oplus Add \oplus Sub$. In the $\gamma\Phi C_0$ ADT definitions, what comes to the r.h.s. of the "=" is called the *case list* of the ADT on the l.h.s. of the "=". The connection between $\gamma\Phi C_0$ and C4C becomes more clear in § 3.1. Hereafter, we refer to α_1 as *NA* (for Numbers and Addition) and to α_2 as *NAM* (for Numbers, Addition, and Multiplication).

2.2 The Scala LMS-EPS

Suppose one is interested in encoding *NA* and in evaluating its expressions. One possible Scala implementation is:

```
1  trait NA {
2    trait Exp                                                  //Exp ::=
3    case class Num(n: Int)          extends Exp                //     Num(n) |
4    case class Add(l: Exp, r: Exp) extends Exp                 //     Add(Exp, Exp)
5    def eval: Exp => Int = {
6      case Num(n)    => n
7      case Add(l, r) => eval(l) + eval(r) } }
```

Scala uses inheritance for definition of ADT cases. In lines 3 and 4 above, for example, `Num` and `Add` inherit from their ADT type, i.e., `Exp`. Implementing *NAM* without manipulation or duplication of `NA` can now be done as:

```
1  trait NAM extends NA {                                  //Exp ::= ... |
2    case class Mul(l: Exp, r: Exp) extends Exp                //     Mul(Exp, Exp)
3    override def eval: Exp => Int = {
4      case Mul(l, r) => eval(l) * eval(r)
5      case e         => super.eval(e)} }
```

Line 2 above adds the new case (`Mul`). Line 4 above handles its evaluation. And, line 5 above makes a **`super`** call to employ the evaluation already defined at `NA`. Note that `NAM` inherits `Num` and `Add` because it **`extends`** `NA`.

Addition of a function on *NA* whilst addressing E3 and E4 is similar. For example, here is how to provide pretty printing:

```
1  trait NAPr extends NA {
2    def to_str: Exp => String = {
3      case Num(n)    => n.toString
4      case Add(l, r) => to_str(l) + " + " + to_str(r)} }
```

3 The C++ Version

C++ offers no built-in support for ADTs. Neither does it support mixin-composition for a `super` call to be possible. The C++ LMS-EPS mitigates those by exercising a coding discipline that is explained in § 3.1 to § 3.3. Term creation and application of functions on that comes in § 3.4.

3.1 Cases

An EP solution takes a C4C approach when each ADT case is implemented using a standalone component that is ADT-parameterised. In the C++ LMS-EPS, the ADT-parametrisation translates into type-parametrisation by ADT. For example, here are the C++ counterparts of `Num` and `Add` in § 2.2:

```
1 template<typename ADT> struct Num {//Num  α : ℤ → α
2   Num(int n): n_(n) {}
3   int n_;
4 };
```

Above comes a C4C equivalent of `Num` in § 2.2. Verbosity aside, an important difference to notice is that `Num` in § 2.2 is a case for the ADT `Exp` of § 2.2, **exclusively**. On the contrary, the above `Num` is a case for the encoding of **every** ADT α such that $Num \in cases(\alpha)$. The `Add` below is similar.

```
1 template<typename ADT> struct Add {//Add  α : α × α → α
2   using CsVar = typename ADT::cases;
3   Add(const CsVar& l, const CsVar& r):
4     l_(std::make_shared<CsVar>(l)), r_(std::make_shared<CsVar>(r)) {}
5   const std::shared_ptr<CsVar> l_, r_;
6 };
```

Terms created using `Add`, however, are recursive w.r.t. their ADT. That is reflected in line 5 with the `l_` and `r_` data members of `Add` being shared pointers to the case list of `ADT`, albeit packed in a `std::variant`. (See line 2 in `NATemp` below.) Line 2 is a type alias that will become more clear in § 3.2. The need for storing `l_` and `r_` in `std::shared_ptr`s is discussed in § 5.1.

We follow the terminology of C++ IDPAM[5] [12] in calling `Num` and `Add` of this section and similar C4C encodings of ADT cases the *case components*.

3.2 ADTs

Defining ADTs in the C++ LMS-EPS is less straightforward:

```
1 template<typename ADT> struct NATemp
2 {using cases = std::variant<Num<ADT>, Add<ADT>>;};
3 struct NA: NATemp<NA> {};
```

In Swierstra's terminology [34], lines 1 and 2 define a recursive knot that line 3 ties. In the terminology of Oliveira et al. [26], `NATemp` is an object algebra interface. That is because `NATemp` declares a set of algebraic signatures (namely, that of `Num` and `Add`) but does not define (implement) them. In other words, those signatures do not pertain to a fixed ADT.

[5] Integration of a Decentralised Pattern Matching

What matters to the C++ LMS-EPS is that `NATemp` underpins every `ADT`, for which instances of `Num<ADT>` or `Add<ADT>` are valid terms. (Using $\gamma\Phi C_0$, one denotes that by $\forall\alpha.\ \alpha \lhd \mathit{Num} \oplus \mathit{App}$.) Given `NATemp`, in line 3, we introduce `NA` as an instance of such `ADT`s. That introduction is done in a specific way for F-Bounding [4] commonly referred to in C++ as the Curiously Recurring Template Pattern (CRTP) [36, §21.2]. See § 5.2 for why employing CRTP is required here.

The nested type name `cases` at line 2 above is what we used in the definition of `CsVar` at line 2 of `Add` in § 3.1.

3.3 Functions

Just like that for ADTs, defining functions on ADTs takes two steps in the C++ LMS-EPS:

First, for a function `f` on an ADT `A`, one implements an auxiliary function that takes a continuation as an argument. Suppose that one chooses the name `a_plus_f_matches` for the auxiliary function. (See below for the intention behind the naming of the auxiliary function.) Using the continuation, `a_plus_f_matches` implements the *raw* pattern matching for `f` on every extension to `A`. Once called with `f` substituted for the continuation, `a_plus_f_matches` returns the pattern matching of `f`, now exclusively *materialised* for `A`.

Second, one implements `f` itself, which, by passing itself to `a_plus_f_matches`, acquires the right pattern matching; and, then, visits `f`'s parameter using the acquired pattern matching.

As an example for the above two steps, we implement below an evaluator for `NA` expressions:

```
1 template<typename ADT> auto na_plus_ev_matches(auto eval)
2 {//na_plus_ev_matches<α ◁ Num ⊕ Add>
3   return match {
4     []    (const Num<ADT>& n) {return n.n_;},                    //λNum(n). n
5                                                                  //λAdd(l,r). eval(l) + eval(r)
6     [eval](const Add<ADT>& a) {return eval(*a.l_) + eval(*a.r_);}
7   };
8 }
```

Above is the first step: `na_plus_ev_matches` is the auxiliary function for evaluation. `eval` in line 1 is the continuation. `na_plus_ev_matches` produces the raw pattern matching for every `ADT` that extends `NA`. It does so by passing match statements for `Num<ADT>` and `Add<ADT>` to the `match` combinator. In line 4, a λ-abstraction is used for matching `Num<ADT>` instances. Line 6, on the other hand, use a λ-abstraction to match `Add<ADT>` instances. The difference is that the latter λ-abstraction is recursive and captures the variable `eval` (by mentioning it between square brackets in line 6). Furthermore, rather than using `na_plus_ev_matches`, it uses the continuation `eval` for recursion.

In short, the `match` combinator bundles a set of match statements together. Such a match statement can be any callable C++ object. In this paper, we only use λ-abstractions for our match statements. More on `match` in § 5.3.

```
1 int na_eval(const NA::cases& expr) {
2   auto pm = na_plus_ev_matches<NA>(na_eval);
3   return std::visit(pm, expr);
4 }
```

Above is the second step for provision of evaluation for `NA` expressions. In line 2, it acquires the right pattern matching for `NA` by passing itself as the continuation to `na_plus_ev_matches`. Then, in line 3, it visits the expression to be evaluated using the acquired pattern matching.

We would like to end this subsection by emphasising on the following: In the terminology of Oliveira et al. [26], `na_plus_ev_matches` is an object algebra. In the latter work, compositionality of object algebras comes at the price of a generalisation of *self*-references [27]. (In short, inside the body of an instance of a given class, a *self*-references is a pointer/reference to the very instance itself. Such a pointer/reference needs to also deal with virtual construction [8].) Notably, however, we achieve that (Section 4.1) without resorting to *self*-references.

3.4 Tests

Using the following two pieces of syntactic sugar for literals and addition

```
auto operator"" _n (unsigned long long n) {return Num<NA>(n);}
auto operator + (const NA::cases& l, const NA::cases& r) {return Add<NA>(l, r);}
```

`na_eval(5_n + 5_n + 4_n)` returns `14`, as expected.

4 Addressing the EP Concerns

We now show how our technology is an EP solution.

4.1 E1 (Bidimensional Extensibility)

Extensibility in the dimension of ADTs is simple. Provided the `Mul` case component below

```
1 template<typename ADT> struct Mul{ //Mul α :: α × α → α
2   using CsVar = typename ADT::cases;
3   Mul(const CsVar& l, const CsVar& r):
4     l_(std::make_shared<CsVar>(l)), r_(std::make_shared<CsVar>(r)) {}
5   const std::shared_ptr<CsVar> l_, r_;
6 };
```

encoding *NAM* using the C++ LMS-EPS can be done just like that for *NA*:

```
1 template<typename ADT> struct NAMTemp
2 {using cases = std::variant<Num<ADT>, Add<ADT>, Mul<ADT>>;};
3 struct NAM: NAMTemp<NAM> {};
```

But, one can also extend `NA` to get `NAM`:

```
1 template<typename ADT> struct NAMTemp
2 {using cases = ext_variant_by_t<NATemp<ADT>, Mul<ADT>>;};
```

In the absence of a built-in **`extends`** for **`trait`**s, that is the C++ LMS-EPS counterpart for extending an ADT to another. See § 5.4 for the definition of `ext_variant_by_t`.

Extensibility in the dimension of functions is not particularly difficult. For example, here is how one does pretty printing for `NA`:

```
1  template<typename ADT> auto na_plus_to_str_matches(auto to_string) {
2    return match {
3      []          (const Num<ADT>& n) {return std::to_string(n.n_);},
4      [to_string](const Add<ADT>& a) {return to_string(*a.l_) + " + " +
5                                                          to_string(*a.r_);}
6    };
7  }
8  std::string na_to_string(const NA::cases& expr) {
9    auto pm = na_plus_to_str_matches<NA>(na_to_string);
10   return std::visit(pm, expr);
11 }
```

`na_plus_to_str_matches` is the auxiliary function with `to_string` being the continuation. `na_to_string` is the pretty printing for `NA`.

```
1  template<typename ADT> auto nam_plus_to_str_matches(auto to_string) {
2    return match {
3      na_plus_to_str_matches<ADT>(to_string),
4      [to_string](const Mul<ADT>& m) {return to_string(*m.l_) + " * " +
5                                                          to_string(*m.r_);}
6    };
7  }
```

On the other hand, the above auxiliary function called `nam_plus_to_str_matches` reuses the match statements already developed by `na_plus_to_str_matches` (line 3). It does so by including the latter function in the list of match statements it includes in its `match` combinator. Note that the former function, moreover, passes its own continuation (i.e., `to_string`) as an argument to the latter function. Such a reuse is the C++ LMS-EPS counterpart of the **`super`** call in line 5 of `NAM` in § 2.2.

The similarity becomes more clear when one observes that both the Scala LMS-EPS and the C++ LMS-EPS scope the match statements and have mechanisms for reusing the existing ones. In the Scala LMS-EPS, the match statements are scoped in a method of the base **`trait`**. That method, then, can be **`override`**n at the extension and reused via a **`super`** call. On the other hand, in the C++ LMS-EPS, the match statements are scoped in the auxiliary functions. That auxiliary function, then, can be mentioned in the `match` of the extension's auxiliary function (just like the new match statements), enabling its reuse.

4.2 E2 (Static Type Safety)

Suppose that in the pretty printing for `NAM`, one mistakenly employs `na_plus_to_str_matches` instead of `nam_plus_to_str_matches`. (Note that the latter name starts with `nam` whilst the former only starts with `na`.) That situation is like when the programmer attempts pretty printing for a `NAM` expression without having provided the pertaining match statement of `Mul`. Here is the erroneous code:

```
1  std::string nam_to_string(const NAM::cases& expr) {//WRONG!
2    auto pm = na_plus_to_str_matches<NAM>(nam_to_string);
3    return std::visit(pm, expr);
4  }
```

As expected, the above code fails to compile. As an example, GCC 7.1 produces three error messages. In summary, those error messages state that

`na_plus_to_str_matches` only has match statements for `Num` and `Add` (but not `Mul`). Note that the code fails to compile even without passing a concrete argument into `nam_to_string`. That demonstrates our **strong** static type safety. The C++ LMS-EPS can guarantee that because the compiler chooses the right match statement using overload resolution, i.e., at compile-time. C.f. § 5.3 for more.

4.3 E3 (No Manipulation/Duplication)

Notice how nothing in the evidence for our support for E1 and E2 requires manipulation, duplication, or recompilation of the existing codebase. Our support for E3 follows.

4.4 E4 (Separate Compilation)

Our support for E4, in fact, follows just like E3. It turns out, however, that C++ **`template`**s enjoy two-phase translation [36, §14.3.1]: Their parts that depend on the type parameters are type checked (and compiled) only when they are instantiated, i.e., when concrete types are substituted for all their type parameters. As a result, type checking (and compilation) will be redone for every instantiation. That type-checking peculiarity might cause confusion w.r.t. our support for E4.

In order to dispel that confusion, we need to recall that `Add`, for instance, is a class **`template`** rather than a class. In other words, `Add` is not a type (because it is of kind $* \rightarrow *$) but `Add<NA>` is. The interesting implication here is that `Add<NA>` and `Add<NAM>` are in no way associated to one another. Consequently, introduction of `NAM` in presence of `NA`, causes no repetition in type checking (or compilation) of `Add<NA>`. (`Add<NAM>`, nonetheless, needs to be compiled in presence of `Add<NA>`.) The same argument holds for every other case component already instantiated with the existing ADTs.

More generally, consider a base ADT $\Phi_b = \oplus\overline{\gamma}$ and its extension $\Phi_e = (\oplus\overline{\gamma}) \oplus (\oplus\overline{\gamma}')$. Let $\#(\overline{\gamma}) = n$ and $\#(\overline{\gamma}') = n'$, where $\#(.)$ is the number of components in the component combination. Suppose a C++ LMS-EPS codebase that contains case components for $\gamma_1, \ldots, \gamma_n$ and $\gamma'_1, \ldots, \gamma'_{n'}$. Defining Φ_b in such a codebase incurs compilation of n case components. Defining Φ_e on top incurs compilation of $n+n'$ case components. Nevertheless, that does not disqualify our EP solution because defining the latter component combination does not incur recompilation of the former component **combination**. Note that individual components differ from their combination. And, E4 requires the combinations not to be recompiled.

Here is an example in terms of DSL embedding. Suppose availability of a type checking phase in a codebase built using the C++ LMS-EPS. Adding a type erasure phase to that codebase, does not incur recompilation of the type checking phase. Such an addition will, however, incur recompilation of the case components common between the two phases. Albeit, those case components will be recompiled for the type erasure phase. That addition leaves the compilation of the same case components for the type checking phase intact. Hence, our support for E4.

A different understanding from separate compilation is also possible, in which: an EP solution is expected to, upon extension, already be done with the type checking and compilation of the "core part" of the new ADT. Consider extending *NA* to *NAM*, for instance. With that understanding, *Num* and *Add* are considered the "core part" of *NAM*. As such, the argument is that the type checking and compilation of that "core part" should not be repeated upon the extension.

However, before instantiating `Num` and `Add` for `NAM`, both `Num<NAM>` and `Add<NAM>` are neither type checked nor compiled. That understanding, hence, refuses to take our work for an EP solution. We find that understanding wrong because the core of *NAM* is *NA*, i.e., the $Num \oplus Add$ **combination**, as opposed to both *Num* and *Add* but individually. Two quotations back our mindset up:

The definition Zenger and Odersky [20] give for separate compilation is as follows: "Compiling datatype extensions or adding new processors should not encompass re-type-checking the original **datatype** or existing processors [functions]." The datatypes here are `NA` and `NAM`. Observe how compiling `NAM` does not encompass repetition in the type checking and compilation of `NA`.

Wang and Oliveira [38] say an EP solution should support: "software evolution in both dimensions in a modular way, without modifying the code that has been written previously." Then, they add: "Safety checks or compilation steps must not be deferred until link or runtime." Notice how neither definition of new case components or ADTs, nor addition of case components to existing ADTs to obtain new ADTs, implies modification of the previously written code. Compilation or type checking of the extension is not deferred to link or runtime either.

For more elaboration on the take of Wang and Oliveira on (bidimensional) modularity, one may ask: If *NA*'s client becomes a client of *NAM*, will the client's code remain intact under E3 and E4? Let us first disregard code that is exclusively written for *NA* for it is not meant for reuse by *NAM*:

```
void na_client_f(const NA&) {...}
```

If on the contrary, the code only counts on the availability of *Num* and *Add*:

```
template <
 typename ADT, typename = std::enable_if_t<adt_contains_v<ADT, Num, Add>>
> void na_plus_client_f(const ADT& x) {...}
```

Then, it can expectedly be reused upon transition from *NA* to *NAM*. (We drop the definition of `adt_contains_v` due to space restrictions.)

5 Technicality

5.1 Why `std::shared_ptr`?

Although not precisely what the C++ specification states, it is not uncommon for the current C++ compilers to require the types participating in the formation of a `std::variant` to be default-constructable. That requirement is, however, not fulfilled by our case components. As a matter of fact, ADT cases, in general, are unlikely to fulfil that requirement.

But, as shown in line 2 of `NATemp`, the C++ LMS-EPS needs the case components to participate in a `std::variant`. Wrapping the case components in a default-constructable type seems inevitable. We choose to wrap them inside a `std::shared_ptr` because, then, we win sub-expression sharing as well.

5.2 Why CRTP?

The reader might have noticed that, in the C++ LMS-EPS, defining ADTs is also possible without CRTP. For example, one might try the following for *NA*:

```
1 struct OtherNA { using cases = std::variant<Num<OtherNA>, Add<OtherNA>>; };
```

Then, extending `OtherNA` to an encoding for *NAM* will, however, not be possible as we extended `NATemp` to `NAMTemp` in § 4.1. In addition to employing a different extension metafunction than `ext_variant_by_t` in § 5.4, we would need some extra work in the case components. For example, here is how to enrich `Add`:

```
1 template<typename ADT> struct Add
2 {/*... like before ... */ template<typename A> using case_component = Add<A>;};
```

Then, we can still extend `NATemp` to get `NAM`:

```
1 struct NAM { using cases = ext_variant_by_t<NATemp<NAM>, Mul<NAM>>; };
```

If one wishes to, it is even possible to completely abolish `NATemp` – and, in fact, all the CRTP:

```
1 struct NAM { using cases = ext_to_by_t<NA, NAM, Mul<NAM>>; };
```

where `ext_to_by_t` is defined in § 5.4.

5.3 The `match` Combinator

The definition of our `match` combinator is as follows[6]:

```
1 template<typename... Ts> struct match: Ts...
2 {using Ts::operator()...;};
3 template<typename... Ts> match(Ts...) -> match<Ts...>;
```

As one can see above, `match` is, in fact, a type parameterised **`struct`**. In lines 1 and 2 above, `match` derives from all its type arguments. At line 2, it also makes all the **`operator`**`()`s of its type arguments accessible via itself. Accordingly, `match` is *callable* in all ways its type arguments are.

Line 3 uses a C++ feature called *user-defined deduction guides*. Recall that C++ only offers automatic type deduction for **`template`** functions. Without line 3, thus, `match` is only a **`struct`**, missing the automatic type deduction. The programmer would have then needed to list all the type arguments explicitly to instantiate `match`. That would have been cumbersome and error-prone – especially, because those types can rapidly become human-unreadable. Line 3 helps

[6] This is a paraphrase of the `overloaded` combinator taken from the `std::visit`'s online specification at the C++ Reference: https://en.cppreference.com/w/cpp/utility/variant/visit

the compiler to deduce type arguments for the `struct` (i.e., the `match` to the right of "->") in the same way it would have done that for the function (i.e., the `match` to the left of "->").

One may wonder why we need all those `Ts::operator ()`s. The reason is that, according to the C++ specification, the first argument of `std::visit` needs to be a *callable*. The compiler tries the second `std::visit` argument against all the call pieces of syntax that the first argument provides. The mechanism is that of C++'s overload resolution. In this paper, we use `match` only for combining λ-abstractions. But, all other sorts of callable are equally acceptable to `match`.

Finally, we choose to call `match` a combinator because, to us, its usage is akin to Haskell's Combinator Pattern [28, §16].

5.4 Definitions of `ext_variant_by_t` and `ext_to_by_t`

Implementation of `ext_variant_by_t` is done using routine `template` metaprogramming:

```
1 template<typename, typename...> struct evb_helper;
2 template<typename... OCs, typename... NCs>
3 struct evb_helper<std::variant<OCs...>, NCs...>
4 {using type = std::variant<OCs..., NCs...>;};
5 template<typename ADT, typename... Cs> struct ext_variant_by
6 {using type = typename evb_helper<typename ADT::cases, Cs...>::type;};
7 template<typename ADT, typename... Cs>
8 using ext_variant_by_t = typename ext_variant_by<ADT, Cs...>::type;
```

`ext_variant_by_t` (line 8) extends an `ADT` by the cases `Cs...`. To that end, `ext_variant_by_t` is a syntactic shorthand for the `type` nested type of `ext_variant_by`. `ext_variant_by` (line 5) works by delegating its duty to `evb_helper` after acquiring the case list of `ADT` (line 6). Given a `std::variant` of old cases (`OCs...`) and a series of new cases (`NCs...`), the metafunction `evb_helper` type-evaluates to a `std::variant` of old and new cases (line 4).

Implementing `ext_to_by_t` is not particularly more complicated. So, we drop explanation and only provide the code:

```
template<typename, typename> struct materialise_for_helper;
template<typename ADT, typename... Cs>
struct materialise_for_helper<ADT, std::variant<Cs...>>
{using type = std::variant<typename Cs::template case_component<ADT>...>;};

template<typename ADT1, typename ADT2> struct materialise_for {
  using type = typename materialise_for_helper<ADT2, typename ADT1::cases>::type;
};

template<typename ADT1, typename ADT2, typename... Cs> struct ext_to_by {
  using type = typename evb_helper<typename materialise_for<ADT1, ADT2>::type,
                                   Cs...>::type;
};

template<typename ADT1, typename ADT2, typename... Cs>
using ext_to_by_t = typename ext_to_by<ADT1, ADT2, Cs...>::type;
```

6 C4C versus GADTs

When embedding DSLs, it is often convenient to piggyback on the host language's type system. In such a practice, GADTs are a powerful means to guarantee the absence of certain type errors. For example, here is a Scala transliteration[7] of the running example Kennedy and Russo [18] give for GADTs in object-oriented languages:

```
1 sealed abstract class Exp[T]
2 case class Lit(i: Int)                                        extends Exp[Int]
3 case class Plus(e1: Exp[Int], e2: Exp[Int])                   extends Exp[Int]
4 case class Equals(e1: Exp[Int], e2: Exp[Int])              extends Exp[Boolean]
5 case class Cond(e1: Exp[Boolean], e2: Exp[Int], e3: Exp[Int]) extends Exp[Int]
6 /* ... more case classes ... */
7 def eval[T](exp: Exp[T]): T = exp match {...}
```

Notice first that `Exp` is type parameterised, where `T` is an arbitrary Scala type. That is how `Lit` can derive from `Exp[Int]` whilst `Equals` derives from `Exp[Boolean]`. Second, note that `Plus` takes two instances of `Exp[Int]`. Contrast that with the familiar encodings of $\alpha = \mathit{Plus}(\alpha, \alpha) \mid \ldots$, for some ADT α. Unlike the GADT one, the latter encoding cannot outlaw nonsensical expressions such as `Plus(Lit(5), Lit(true))`. Third, note that `eval` is polymorphic in the carrier type of `Exp`, i.e., `T`.

The similarity between the above case definitions and our case components is that they are both type parameterised. Nevertheless, the former are parameterised by the type of the Scala expression they carry. Whereas, our case components are parameterised by their ADT types. The impact is significant. Suppose, for example, availability of a case component `Bool`with the corresponding `operator ""_b` syntactic sugar. In their current presentation, our case components cannot statically outlaw type-erroneous expressions like `12_n + "true"_b`. On the other hand, the GADT `Cond` is exclusively available as an ADT case of `Exp` and cannot be used for other ADTs.

Note that, so long as statically outlawing `12_n + "true"_b` is the concern, one can always add another layer in the `Exp` grammar so that the integral cases and Boolean cases are no longer at the exact same ADT. That workaround, however, will soon become unwieldy. That is because, it involves systematically separating syntactic categories for every carrier type – resulting in the craft of a new type system. GADTs employ the host language's type system instead.

The bottom line is that GADTs and C4C encodings of ADTs are orthogonal. One can always generalise our case components so they too are parameterised by their carrier types and so they can guarantee similar type safety.

7 Related Work

The support of the Scala LMS-EPS for E2 can be easily broken using an incomplete pattern matching. Yet, given that Scala pattern matching is dynamic,

[7] Posted online by James Iry on Wed, 22/10/2008 at http://lambda-the-ultimate.org/node/1134.

whether LMS really relaxes E2 is debatable. Note that the problem in the Scala LMS-EPS is not an "Inheritance is not Subtyping" one [7]: The polymorphic function of a deriving `trait` does specialise that of the base.

In comparison to the Scala LMS-EPS, we require one more step for defining ADTs: the CRTP. Nevertheless, given that the C++ LMS-EPS is C4C, specifying the cases of an ADT is by only listing the right case components in a `std::variant`. Defining functions on ADTs also requires one more step in the C++ LMS-EPS: using the continuation. When extending a function for new ADT cases, the C++ LMS-EPS, however, needs no explicit `super` call, as required by the Scala LMS-EPS.

A note on the Expression Compatibility Problem is appropriate here. As detailed earlier [11, § 4.2], the Scala LMS-EPS cannot outlaw incompatible extensions. Neither can the current presentation of the C++ LMS-EPS. Nonetheless, due to its C4C nature, that failure is not inherent in the C++ LMS-EPS. One can easily constrain the `ADT` type parameter of the case components in a similar fashion to the Scala IDPAM [16] to enforce compatibility upon extension.

The first C4C solution to the EP is the Scala IDPAM [16]. ADT creation in the Scala IDPAM too requires F-Bounding. But, the type annotation required when defining an ADT using their case components is heavier.

In the Scala IDPAM, the number of type annotations required for a function taking an argument of an ADT with n cases is $\mathcal{O}(n)$. That is $\mathcal{O}(1)$ in the C++ LMS-EPS. The reason is that, in C++, with programming purely at the type level, types can be computed from one another. In particular, an ADT's case list can be computed programmatically from the ADT itself. That is not possible in Scala without `implicit`s, which are not always an option. In the Scala IDPAM too, implementation of functions on ADTs is *nominal*: For every function on a given ADT α, all the corresponding match components—i.e., match statements also delivered as components— of α's cases need to be manually mixed in to form the full function implementation. The situation is similar for the C++ LMS-EPS in that all the match statements are required to be manually listed in the `match` combinator. However, instead of using a continuation, in the Scala IDPAM, one mixes in a base case as the last match component. Other than F-Bounding, the major language feature required for the Scala IDPAM is stackability of `trait`s. In the C++ LMS-EPS, that is variadic `template`s. The distinctive difference between the C++ LMS-EPS and the Scala IDPAM is that the latter work relaxes E2 in the absence of a default [39]. On the contrary, the C++ LMS-EPS guarantees strong static type safety.

The second C4C solution to the EP is the C++ IDPAM [12]. There are two reasons to prefer the C++ IDPAM over the C++ LMS-EPS: Firstly, in the C++ IDPAM, definition of a function f on ADTs amounts to provision of simple (C++) function overloads, albeit plus a one-off macro instantiation for f. (Those function overloads are called match components of the C++ IDPAM.) Secondly, in the C++ IDPAM, function definition is *structural*: Suppose the availability of all the corresponding match components of α's case list and the macro instantiation for f. Then, unlike the C++ LMS-EPS, to define f on α, the

programmer need not specify which match statements to include in the pattern matching. The compiler deductively obtains the right pattern matching using α's structure, i.e. α's case list.

There are two reasons to prefer the C++ LMS-EPS over the C++ IDPaM. Firstly, implementing ADTs and functions on them is only possible in the C++ IDPaM using a metaprogramming facility shipped as a library. That library was so rich in its concepts that it was natural to extend [13] for multiple dispatch. Behind the scenes, the library performs iterative pointer introspection to choose the right match statements. In the C++ LMS-EPS, that pointer introspection is done using the compiler's built-in support for `std::variant`. That saves the user from having to navigate the metaprogramming library upon mistakes (or bugs). Furthermore, when it comes to orchestrating the pattern matching, the compiler is likely to have more optimisation opportunities than the library. Secondly, unlike their C++ IDPaM equivalents, case components of the C++ LMS-EPS do not inherit from their ADT. This entails weaker coupling between case components and ADT definitions.

Instead of `std::variant`, one can use `boost::variant`[8] to craft a similar solution to the C++ LMS-EPS. Yet, the solution would have not been as clean with its auxiliary functions as here. In essence, for a function f, one would have needed to manually implement each match statement as a properly-typed overload of `F::operator ()`. Extending f to handle new ADT cases, nevertheless, would have been more akin to the Scala LMS-EPS. That is because, then, providing the new match statements would have amounted to implementing the corresponding `FExtended::operator ()` overloads, for some `FExtended` that derives from `F`. (Compare with § 2.2.) Moreover, `boost::variant` requires special settings for working with recursive types (such as ADTs) that damage readability.

Using object algebras [10] to solve EP has become popular over recent years. Oliveira and Cook [23] pioneered that. Oliveira et al. [26] address some awkwardness issues faced upon composition of object algebras. Rendel, Brachthäuser and Ostermann [30] add ideas from attribute grammars to get reusable tree traversals. As also pointed out by Black [3], an often neglected factor about solutions to EP is the complexity of term creation. That complexity increases from one work to the next in the above literature. The symptom develops to the extent that it takes Rendel, Brachthäuser and Ostermann 12 non-trivial lines of code to create a term representing "3+5". Of course, those 12 lines are not for the latter task exclusively and enable far more reuse. Yet, those 12 lines are inevitable for term creation for "3+5", making that so heavyweight. The latter work uses automatic code generation for term creation. So, the ADT user has a considerably more involved job using the previous object algebras technologies for EP than that of ours. Additionally, our object algebras themselves suffer from much less syntactic noise. Defining functions on ADTs is slightly more involved in the C++ LMS-EPS than object algebras for the EP. For example, pretty-printing for *NA*

[8] https://www.boost.org/doc/libs/1_67_0/doc/html/variant.html

takes 12 (concise) Scala lines in the latter work, whereas that is 14 (syntactically noisy) C++ lines in ours.

Garrigue [9] solves EP using global case definitions that, at their point of definition, become available to every ADT defined afterwards. Per se, a function that pattern matches on a group of these global cases can serve any ADT containing the selected group. OCaml's built-in support for Polymorphic Variants [9] makes definition of both ADTs and functions on them easier. However, we minimise the drawbacks [3] of ADT cases being global by promoting them to components.

Swierstra's Datatypes à la Carte [34] uses Haskell's type classes to solve EP. In his solution too, ADT cases are ADT-independent but ADT-parameterised. He uses Haskell Functors to that end. Defining functions on ADTs amounts to defining a type class, instances of which materialising match statements for their corresponding ADT cases. Without syntactic sugaring, term creation can become much more involved than that for ordinary ADTs of Haskell. Defining the syntactic sugar takes many more steps than us, but, makes term creation straightforward. Interestingly enough, using the Scala type classes [24] can lead to simpler syntactic sugar definition but needs extra work for the lack of direct support in Scala for type classes. In his machinery, Swierstra offers a `match` that is used for monadically inspecting term structures.

Bahr and Hvitved extend Swierstra's work by offering Compositional Datatypes (CDTs) [1]. They aim at higher modularity and reusability. CDTs support more recursion schemes, and, extend to mutually recursive data types and GADTs. Besides, syntactic sugaring is much easier using CDTs because smart constructors can be automatically deduced for terms.

Later on, they offer Parametric CDTs (PCDTs) [2] for automatic α-equivalence and capture-avoiding variable bindings. PCDTs achieve that using Difunctors [19] (instead of functors) and a CDT encoding of Parametric Higher-Order Abstract Syntax [5]. Case definitions take two phases: First an equivalent of our case components need to be defined. Then, their case components need to be materialised for each ADT, similar to but different from that of Haeri and Schupp [14,11].

The distinctive difference between C4C and the works of Swierstra, Bahr, and Hvitved is the former's inspiration by CBSE. Components, in their CBSE sense, ship with their 'requires' and 'provides' interfaces. Whereas, even though the latter works too parametrise cases by ADTs, the interface that CDTs, for instance, define do not go beyond algebraic signatures. Although we do not present those for C++ LMS-EPS here, C4C goes well beyond that, enabling easy solutions to the Expression Families Problem [23] and Expression Compatibility Problem [16] as well as GADTs. The respective article is in submission.

8 Conclusion

In this paper we show how a new C4C encoding of ADTs in C++ can solve EP in a way that is reminiscent to the Scala LMS-EPS. On its way, our solution

gives rise to simple encodings for object algebras and object algebra interfaces and relates to Datatypes à la Carte ADT encodings.

Given the simplicity of our encoding for object algebras and object algebra interfaces in the absence of heavy notation for term creation, an interesting future work is mimicking the earlier research on object algebra encodings for EP. We need to investigate whether our technology still remains simple when we take all the challenges those works take. Another possible future work is extension of our (single dispatch) mechanism for implementing functions on ADTs to multiple dispatch. Of course, C++ LMS-EPS needs far more experimentation with real-size test cases to study its scalability. Finally, we are working on a C++ LMS-EPS variation that, unlike our current presentation, structurally implements functions on ADTs. The latter variation has thus far presented itself as a promising vehicle for also delivering multiple dispatch.

References

1. P. Bahr and T. Hvitved. Compositional Data Types. In J. Järvi and S.-C. Mu, editors, 7^{th} *WGP*, pages 83–94, Tokyo, Japan, September 2011. ACM.
2. P. Bahr and T. Hvitved. Parametric Compositional Data Types. In J. Chapman and P. B. Levy, editors, 4^{th} *MSFP*, volume 76 of *ENTCS*, pages 3–24, February 2012.
3. A. P. Black. The Expression Problem, Gracefully. In M. Sakkinen, editor, *MASPEGHI@ECOOP 2015*, pages 1–7. ACM, July 2015.
4. P. Canning, W. R. Cook, W. Hill, W. Olthoff, and J. C. Mitchell. F-Bounded Polymorphism for Object-Oriented Programming. In 4^{th} *FPCA*, pages 273–280, September 1989.
5. A. Chlipala. Parametric Higher-Order Abstract Syntax for Mechanized Semantics. In J. Hook and P. Thiemann, editors, 13^{th} *ICFP*, pages 143–156, Victoria, BC, Canada, September 2008.
6. W. R. Cook. Object-Oriented Programming Versus Abstract Data Types. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *FOOL*, volume 489 of *LNCS*, pages 151–178, Noordwijkerhout (Holland), June 1990.
7. W. R. Cook, W. L. Hill, and P. S. Canning. Inheritance is not Subtyping. In 17^{th} *POPL*, pages 125–135, San Francisco, CA, USA, 1990. ACM.
8. E. Ernst, K. Ostermann, and W. R. Cook. A Virtual Class Calculus. In J. G. Morrisett and S. L. Peyton Jones, editors, 33^{rd} *POPL*, pages 270–282. ACM, January 2006.
9. J. Garrigue. Code Reuse through Polymorphic Variants. In *FSE*, number 25, pages 93–100, 2000.
10. J. V. Guttag and J. J. Horning. The Algebraic Specification of Abstract Data Types. *Acta Informatica*, 10:27–52, 1978.
11. S. H. Haeri. *Component-Based Mechanisation of Programming Languages in Embedded Settings*. PhD thesis, STS, TUHH, Germany, December 2014.
12. S. H. Haeri and P. W. Keir. Metaprogramming as a Solution to the Expression Problem. available online, November 2019.
13. S. H. Haeri and P. W. Keir. Multiple Dispatch using Compile-Time Metaprogramming. Submitted to $16^{th} ICTAC$, November 2019.

14. S. H. Haeri and S. Schupp. Reusable Components for Lightweight Mechanisation of Programming Languages. In W. Binder, E. Bodden, and W. Löwe, editors, 12^{th} *SC*, volume 8088 of *LNCS*, pages 1–16. Springer, June 2013.
15. S. H. Haeri and S. Schupp. Expression Compatibility Problem. In J. H. Davenport and F. Ghourabi, editors, 7^{th} *SCSS*, volume 39 of *EPiC Comp.*, pages 55–67. EasyChair, March 2016.
16. S. H. Haeri and S. Schupp. Integration of a Decentralised Pattern Matching: Venue for a New Paradigm Intermarriage. In M. Mosbah and M. Rusinowitch, editors, 8^{th} *SCSS*, volume 45 of *EPiC Comp.*, pages 16–28. EasyChair, April 2017.
17. C. Hofer, K. Ostermann, T. Rendel, and A. Moors. Polymorphic Embedding of DSLs. In Y. Smaragdakis and J. G. Siek, editors, 7^{th} *GPCE*, pages 137–148, Nashville, TN, USA, October 2008. ACM.
18. A. Kennedy and C. V. Russo. Generalized Algebraic Data Types and Object-Oriented Programming. In R. E. Johnson and R. P. Gabriel, editors, 20^{th} *OOPSLA*, pages 21–40, San Diego, CA, USA, October 2005. ACM.
19. E. Meijer and G. Hutton. Bananas in Space: Extending Fold and Unfold to Exponential Types. In J. Williams, editor, 7^{th} *FPCA*, pages 324–333, La Jolla, California, USA, June 1995. ACM.
20. M. Odersky and M. Zenger. Independently Extensible Solutions to the Expression Problem. In *FOOL*, January 2005.
21. M. Odersky and M. Zenger. Scalable Component Abstractions. In 20^{th} *OOPSLA*, pages 41–57, San Diego, CA, USA, 2005. ACM.
22. B. C. d. S. Oliveira. Modular Visitor Components. In 23^{rd} *ECOOP*, volume 5653 of *LNCS*, pages 269–293. Springer, 2009.
23. B. C. d. S. Oliveira and W. R. Cook. Extensibility for the Masses – Practical Extensibility with Object Algebras. In 26^{th} *ECOOP*, volume 7313 of *LNCS*, pages 2–27. Springer, 2012.
24. B. C. d. S. Oliveira, A. Moors, and M. Odersky. Type Classes as Objects and Implicits. In W. R. Cook, S. Clarke, and M. C. Rinard, editors, 25^{th} *OOPSLA*, pages 341–360. ACM, October 2010.
25. B. C. d. S. Oliveira, S.-C. Mu, and S.-H. You. Modular Reifiable Matching: A List-of-Functors Approach to Two-Level Types. In B. Lippmeier, editor, 8^{th} Haskell, pages 82–93. ACM, September 2015.
26. B. C. d. S. Oliveira, T. van der Storm, A. Loh, and W. R. Cook. Feature-Oriented Programming with Object Algebras. In Giuseppe Castagna, editor, 27^{th} *ECOOP*, volume 7920 of *LNCS*, pages 27–51, Montpellier, France, 2013. Springer.
27. K. Ostermann. Dynamically Composable Collaborations with Delegation Layers. In B. Magnusson, editor, 16^{th} *ECOOP*, volume 2374 of *LNCS*, pages 89–110. Springer, June 2002.
28. B. O'Sullivan, J. Goerzen, and D. Stewart. *Real World Haskell: Code You Can Believe in.* O'Reilly, 2008.
29. R. S. Pressman. *Software Engineering: A Practitioner's Approach.* McGraw-Hill, 7^{th} edition, 2009.
30. T. Rendel, J. I. Brachthäuser, and K. Ostermann. From Object Algebras to Attribute Grammars. In A. P. Black and T. D. Millstein, editors, 28^{th} *OOPSLA*, pages 377–395. ACM, October 2014.
31. J. C. Reynolds. User-Defined Types and Procedural Data Structures as Complementary Approaches to Type Abstraction. In S. A. Schuman, editor, *New Direc. Algo. Lang.*, pages 157–168. INRIA, 1975.

32. T. Rompf and M. Odersky. Lightweight Modular Staging: a Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In 9^{th} *GPCE*, pages 127–136, Eindhoven, Holland, 2010. ACM.
33. I. Sommerville. *Software Engineering*. Addison-Wesley, 9^{th} edition, 2011.
34. W. Swierstra. Data Types à la Carte. *JFP*, 18(4):423–436, 2008.
35. M. Torgersen. The Expression Problem Revisited. In M. Odersky, editor, 18^{th} *ECOOP*, volume 3086 of *LNCS*, pages 123–143, Oslo (Norway), June 2004.
36. D. Vandevoorde, N. M. Josuttis, and D. Gregor. *C++ Templates: The Complete Guide*. Addison Wesley, 2^{nd} edition, 2017.
37. P. Wadler. The Expression Problem. Java Genericity Mailing List, November 1998.
38. Y. Wang and B. C. d. S. Oliveira. The Expression Problem, Trivially! In 15^{th} *Modularity*, pages 37–41, New York, NY, USA, 2016. ACM.
39. M. Zenger and M. Odersky. Extensible Algebraic Datatypes with Defaults. In 6^{th} *ICFP*, pages 241–252, Florence, Italy, 2001. ACM.

A C++ Features Used

A C++ `struct` (or `class`) can be type parameterised. The `struct S` below, for example, takes two type parameters `T1` and `T2`:

```
template<typename T1, typename T2> struct S {...};
```

Likewise, C++ functions can take type parameters:

```
template<typename T1, typename T2> void f(T1 t1, T2 t2) {...}
```

From C++20 onward, certain type parameters need not to be mentioned explicitly. For example, the above function `f` can be *abbreviated* as:

```
void f(auto t1, auto t2) {...}
```

A (`template` or non-`template`) `struct` can define nested type members. For example, the `struct T` below defines `T::type` to be `int`:

```
struct T {using type = int;};
```

Nested types can themselves be type parameterised, like `Y::template type`:

```
struct Y {template<typename> using type = int;};
```

C++17 added `std::variant` as a type-safe representation for unions. An instance of `std::variant`, at any given time, holds a value of one of its alternative types. That is, the static type of such an instance is that of the `std::variant` it is defined with; whilst, the dynamic type is one and only one of those alternative types. As such, a function that is to be applied on a `std::variant` needs to be applicable to its alternative types. Technically, a *visitor* is required for the alternative types. The function `std::visit`, takes a visitor in addition to a pack of arguments to be visited.

```
auto twice = [](int n){return n * 2;}
```

The variable `twice` above is bound to a λ-abstraction that, given an `int`, returns its value times two. λ-abstractions can also capture unbound names. In such a case, the captured name needs to be mentioned in the opening square brackets before the list of parameters. For example, the λ-abstraction `times` below captures the name `m`:

```
auto times = [m](int n){return n * m;}
```